AFRL-IF-RS-TR-2001-111
Final Technical Report
June 2001

ON BUILDING A SCALABLE REAL-TIME FAULT-TOLERANT SYSTEM FOR EMBEDDED APPLICATIONS

University of Michigan

Sponsored by
Defense Advanced Research Projects Agency
DARPA Order No. D341

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

AFRL-IF-RS-TR-2001-111 has been reviewed and is approved for publication.

APPROVED: [illegible]

THOMAS F. LAWRENCE
Project Engineer

FOR THE DIRECTOR: [illegible]

WARREN H. DEBANY, Technical Advisor
Information Grid Division
Information Directorate

ON BUILDING A SCALABLE REAL-TIME FAULT-TOLERANT SYSTEM FOR EMBEDDED APPLICATIONS

T. Abdelzaher, S. Dawson,
W. C. Feng, F. Jahanian,
S. Iekel-Johnson, A. Mehra, T. Mitton,
A. Shaikh, Kang G. Shin,
A. Wang, and H. Zou

Contractor:	University of Michigan
Contract Number:	F30602-95-1-0044
Effective Date of Contract:	15 September 1995
Contract Expiration Date:	31 August 2000
Short Title of Work:	On Building A Scalable Real-Time Fault-Tolerant System For Embedded Applications
Period of Work Covered:	Sep 95 - Aug 00
Principal Investigator:	Kang G. Shin
Phone:	(313) 763-0391
AFRL Project Engineer:	Thomas F. Lawrence
Phone:	(315) 330-2925

This research was supported by the Defense Advanced Research Projects Agency of the Department of Defense and was monitored by Thomas F. Lawrence, AFRL/IFGA, 525 Brooks Road, Rome, NY.

REPORT DOCUMENTATION PAGE	*Form Approved* *OMB No. 0704-0188*

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE JUNE 2001	3. REPORT TYPE AND DATES COVERED Final Sep 95 - Aug 00

4. TITLE AND SUBTITLE ON BUILDING A SCALABLE REAL-TIME FAULT-TOLERANT SYSTEMS FOR EMBEDDED APPLICATIONS	5. FUNDING NUMBERS C - F30602-95-1-0044 PE - 62301E PR - C929 TA - 02 WU - 09
6. AUTHOR(S) T. Abdelzaher, S. Dawson, W. C. Feng, F. Jahanian, S. Iekel-Johnson, A. Mehra, T. Mitton, A. Shaikh, Kang G. Shin, A. Wang, and H. Zou	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) University of Michigan Department of Electrical Engineering and Computing Science Real-Time Computing Laboratory Ann Arbor Michigan 48109-2122	8. PERFORMING ORGANIZATION REPORT NUMBER N/A
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Defense Advanced Research Projects Agency 3701 North Fairfax Drive Arlington Virginia 22203-1714 Air Force Research Laboratory 525 Brooks Road Rome New York 13441-4505	10. SPONSORING/MONITORING AGENCY REPORT NUMBER AFRL-IF-RS-TR-2001-111

11. SUPPLEMENTARY NOTES
Air Force Research Laboratory Project Engineer: Thomas F. Lawrence/IFGA/(315) 330-2925

12a. DISTRIBUTION AVAILABILITY STATEMENT APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.	12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

Real-time embedded systems have evolved during the past several decades from small custom-designed digital hardware to large distributed processing systems. As these systems become more complex, their interoperability, evolvability and cost-effectiveness requirements motivate the use of the commercial-off-the-shelf components. This raises the challenge of constructing dependable and predictable real-time services for application developers on top of the inexpensive hardware and software components which has minimal support for timeliness and dependability guarantees We are addressing this challenge in the ARMADA project.

ARMADA is a set of communication and middleware services that provide support for fault-tolerance and end-to-end guarantees for embedded real-time distributed applications. Since real-time performance of such applications depends heavily on the communication subsystem, the first thrust of the project is to develop a predictable communication service and architecture to ensure QoS-sensitive message delivery. In its second thrust, ARMADA aims to offload the complexity of developing fault-tolerance applications from the application programmer by focusing on a collection of modular, composable middleware for fault-tolerance group communication and replication under timing constraints. Finally, we develop tools for testing and validating the behavior of our services.

14. SUBJECT TERMS Real-Time, Embedded Distributed Processing, Fault-Tolerance, Quality of Service, Middleware, Group Communication	15. NUMBER OF PAGES 36
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

TABLE OF CONTENTS

LIST OF FIGURES

1 Introduction

ARMADA is a collaborative project between the Real-Time Computing Laboratory (RTCL) at the University of Michigan and the Honeywell Technology Center. The goal of the project is to develop and demonstrate an integrated set of communication and middleware services and tools necessary to realize embedded fault-tolerant and real-time services on distributed, evolving computing platforms. These techniques and tools together compose an environment of capabilities for designing, implementing, modifying, and integrating real-time distributed systems. Key challenges addressed by the ARMADA project include: timely delivery of services with end-to-end soft/hard real-time constraints; dependability of services in the presence of hardware or software failures; scalability of computation and communication resources; and exploitation of open systems and emerging standards in operating systems and communication services.

ARMADA communication and middleware services are motivated by the requirements of large embedded applications such as command and control, automated flight, shipboard computing, and radar data processing. Traditionally, such embedded applications have been constructed from special-purpose hardware and software. This approach results in high production cost and poor interoperability making the system less evolvable and more prone to local failures. A recent trend, therefore, has been to build embedded systems using Commercial-Off-The-Shelf (COTS) components such as PC boards, Ethernet links, and PC-based real-time operating systems. This makes it possible to take advantage of available development tools, leverage on mass production costs, and make better use of component interoperability. From a real-time application developer's point of view, the approach creates the need for generic high-level software services that facilitate building embedded distributed real-time applications on top of inexpensive widely available hardware. Real-time operating systems typically implement elementary subsets of real-time services. However, monolithically embedding higher-level support in an operating system kernel is not advisable. Different applications have different real-time and fault-tolerance requirements. Thus, catering to all possible requirement ranges in a single operating system would neither be practical nor efficient. Instead, we believe that a composable set of services should be developed of which only a subset may need to exist for any given application. This philosophy advocates the use of a real-time microkernel equipped with basic real-time support such as priority-based scheduling and real-time communication, in addition to a reconfigurable set of composable middleware layered on top of the kernel. Appropriate testing and validation tools should be independently developed to verify required timeliness and fault-tolerance properties of the distributed middleware.

The ARMADA project is therefore divided into three complementary thrust areas: (i) low-level real-time communication support, (ii) middleware services for group communication and fault-tolerance, and (iii) dependability evaluation and validation tools. Figure 1 summarizes the structuring of the ARMADA environment.

The first thrust focused on the design and development of real-time communication services for a microkernel. A generic architecture is introduced for designing the communication subsystem on hosts so that predictability and QoS guarantees are maintained. The architecture is independent of the particular communication service. It is illustrated in this paper in the context of presenting the design of the *real-time channel*; a low-level communication service that implements a simplex, ordered virtual connection between two networked hosts that provides deterministic or statistical end-to-end delay guarantees between a sender-receiver pair.

The second thrust of the project has focused on a collection of modular and composable middleware services (or building blocks) for constructing embedded applications. A layered open-architecture supports modular insertion of a new service or implementation as requirements evolve over the life-span of a system. The ARMADA middleware services include a suite of fault-tolerant group communication services with real-time guarantees, called RTCAST, to support embedded applications with fault-tolerance and timeliness requirements. RTCAST consists of a collection of middleware including a group membership service, a timed atomic multicast service, an admission control and schedulability module, and a clock synchronization service. The ARMADA middleware

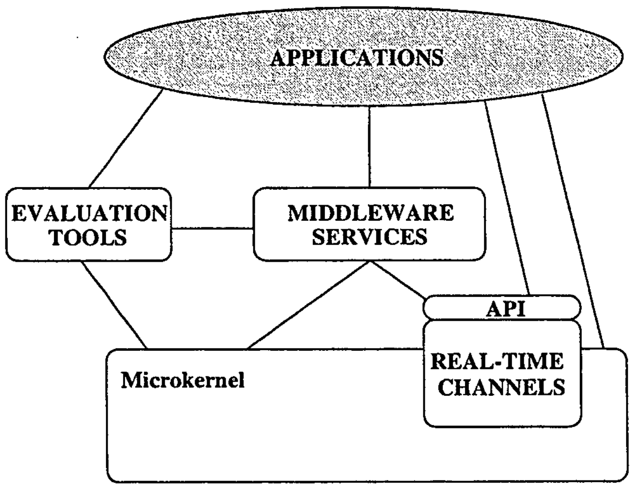


Figure 1: Overview of ARMADA Environment.

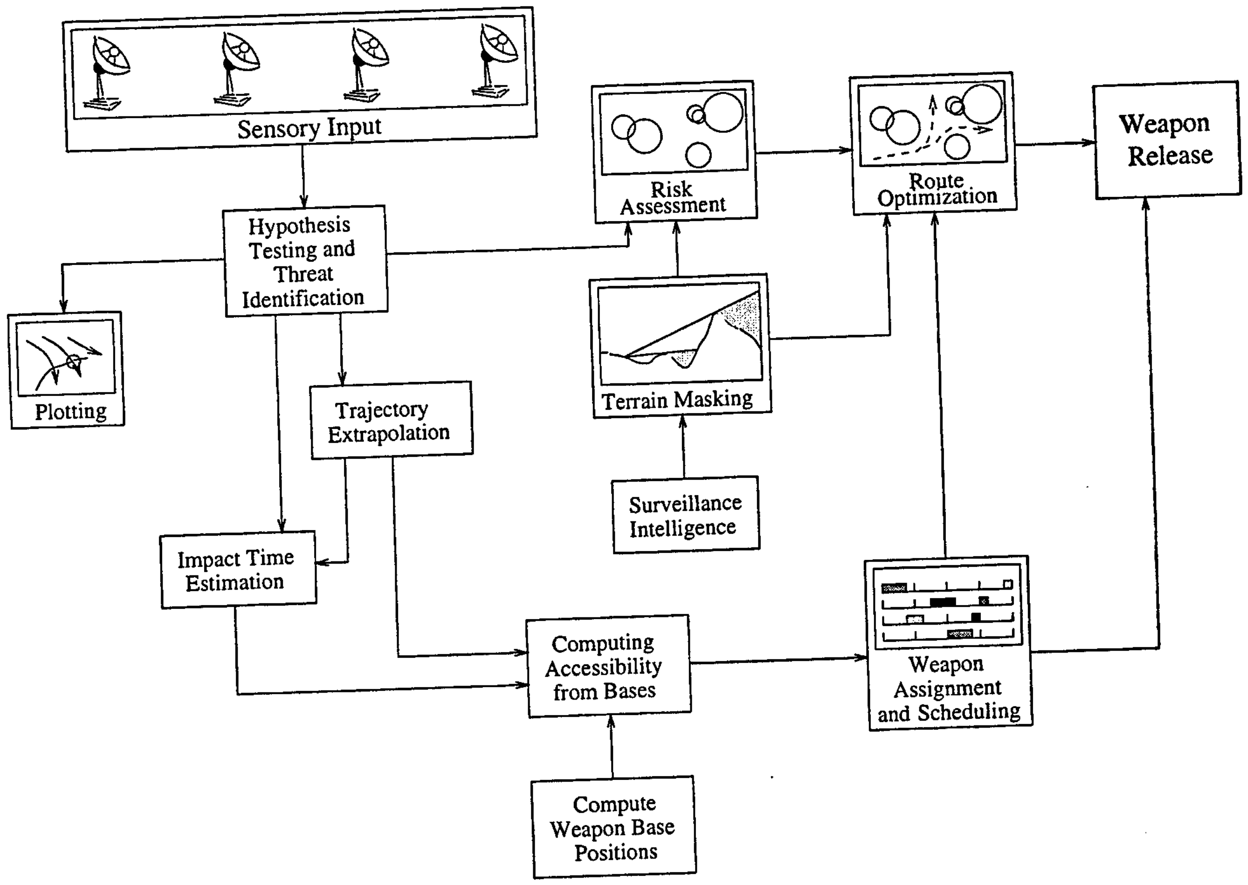
services also include a real-time primary-backup replication service, called RTPB, which ensures *temporally consistent* replicated objects on redundant nodes.

The third thrust of the project is to build a toolset for validating and evaluating the timeliness and fault-tolerance capabilities of the target system. Tools under development include fault injectors at different levels (e.g. operating system, communication protocol, and application), a synthetic real-time workload generator, and a dependability/performance monitoring and visualization tool. The focus of the toolset research is on portability, flexibility, and usability.

Figure 2 gives an overview of a prospective application to illustrate the utility of our services for embedded real-time fault-tolerant systems. The application, developed at Honeywell, is a subset of a command and control facility. Consider a radar installation where a set of sensors are used to detect incoming threats (e.g., enemy planes or missiles in a battle scenario); hypotheses are formed regarding the identity and positions of the threats, and their flight trajectories are computed accordingly. These trajectories are extrapolated into the future and deadlines are imposed to intercept them. The time intervals during which the estimated threat trajectories are reachable from various ground defense bases are estimated; and appropriate resources (weapons) are committed to handle the threats; eventually, the weapons are released to intercept the threats.

The services required to support writing such applications come naturally from their operating requirements. For example, for the anticipated system load, communication between different system components (the different boxes in Figure 2) must occur in bounded time to ensure a bounded end-to-end response from threat detection to weapon release. Our real-time communication services compute and enforce predictable deterministic bounds on message delays given application traffic specification. Critical system components such as hypothesis testing and threat identification have high dependability requirements which are best met using active replication. For such components, RTCAST exports multicast and membership primitives to facilitate fault detection, fault handling, and consistency management of actively replicated tasks. Similarly, extrapolated trajectories of identified threats represent critical system state. A backup of such state needs to be maintained continually and updated to represent the current state within a tolerable consistency (or error) margin. Our primary-backup replication service is implemented to meet such temporal consistency requirements. Finally, our testing tools decrease development and debugging costs of the distributed application.

The rest of this paper is organized as follows. Section `II:` :output lists the students who participated in the project, and contains a list of publications based on work that was conducted as part of the Armada research



Sensory Input
Hypothesis Testing and Threat Identification
Plotting
Trajectory Extrapolation
Impact Time Estimation
Risk Assessment
Route Optimization
Weapon Release
Terrain Masking
Surveillance Intelligence
Computing Accessibility from Bases
Weapon Assignment and Scheduling
Compute Weapon Base Positions

Figure 2: A command and control application

program. Section 3 describes the general approach for integrating ARMADA services into a microkernel framework. It also presents the experimental testbed and implementation environment of this project. The subsequent sections focus on the architecture, design, and implementation of key communication and middleware services in ARMADA. Section 4 introduces real-time communication service. Section 5 presents the RTCAST suite of group communication and fault-tolerance services. Section 6 describes the RTPB (real-time primary-backup) replication service. Section 7 briefly discusses the dependability evaluation and validation tools developed in this project. Section 8 concludes the paper.

2 Participants and Publications

The research efforts conducted as part of the Armada project involved a number of graduate students from the University of Michigan, as well as research staff from the University of Michigan and Honeywell Technology Center. The primary research staff included:

1. Kang Shin (Co-Principal Investigator), Univ. of Michigan
2. Farnam Jahanian (Co-Principal Investigator), Univ. of Michigan
3. Peter Chen, Univ. of Michigan
4. Sunondo Ghosh, Honeywell Technology Center
5. Brian Vanvoorst, Honeywell Technology Center
6. Nicholas Weininger, Honeywell Technology Center
7. Jeremy Norton, Honeywell Technology Center

Students who were awarded PhDs wholly or in part based on their work for the Armada project include:

1. Tarek Abdelzaher
2. Scott Dawson
3. Wu-Chang Feng
4. Scott Iekel-Johnson
5. Atri Indiresan
6. Ashish Mehra
7. Anees Shaikh
8. Hengming Zou

The remaining students, and any degrees they received from the project, are:

1. J.-J. Liou (MS)
2. Pedro Marron (MS)

3. Todd Mitton (MS)

4. Ziqun Wang (MS)

Results from the Armada project have been published in a number of technical journals, conferences, and workshops. They provide a more detailed description of our research results than can be reasonably contained in this summary report. These publications include:

1. T. Abdelzaher, E. Atkins, and K. G. Shin, "QoS negotiation in real-time systems and its application to automated flight control," *IEEE Trans. on Computers*, vol. 49, no. 11, pp. 1155–1169, Nov. 2000.

2. S. Johnson, F. Jahanian, A. Miyoshi, D. de Niz, and R. Rajkumar, "Constructing Real-time Group Communication Middleware Using the Resource Kernel," *Proc. 21st IEEE Real-Time Systems Symposium (RTSS-2000)*, Orlando, FL, November, 2000, pp. 3-12.

3. S. Johnson, F. Jahanian, S. Ghosh, B. Vanvoorst, N. Weininger, and W. Heimerdinger, "Experiences with Group Communication Middleware," Practical Experience Report, *Proc. IEEE International Conference on Dependable Systems and Networks 2000 (FTCS-30/DCCA-8)*, New York, New York, June, 2000, pp. 3-12.

4. T. F. Abdelzaher and K. G. Shin, "Period-based load partitioning and assignment for large real-time applications," *IEEE Trans. on Computers*, vol. 49, no. 1, pp. 81–87, January 2000.

5. Tarek Abdelzaher and Kang G. Shin, "QoS provisioning with *q*Contracts in web and multimedia servers," *Proc. IEEE RTSS99*, pp. 44–53, Phoenix, AZ, Dec. 1999.

6. Tarek F. Abdelzaher and K. G. Shin, "Combined task and message scheduling in distributed real-time systems," *IEEE Trans. on Parallel and Distributed Systems*, vol. 10, no. 11, pp. 1179–1191, Nov. 1999.

7. T. Abdelzaher, S. Dawson, W.-C. Feng, F. Jahanian, S. Johnson, A. Mehra, T. Mitton, A. Shaikh, K. Shin, Z. Wang, and H. Zou, "ARMADA Middleware and Communication Services," *Real-Time Systems Journal*, v. 16(2-3), pp. 127-153, 1999.

8. Scott Johnson, Farnam Jahanian, and Jigney Shah, "The Inter-Group Router Approach to Scalable Group Composition," *Proc. of the 19th International Conference on Distributed Computing Systems (ICDCS99)*, Austin, TX, pp. 4-14, June, 1999.

9. H. Zou and F. Jahanian, "Real-Time Primary-Backup Replication with Temporal Consistency Guarantees," *IEEE Transactions on Parallel & Distributed Systems*, vol. 10, no. 6, pp. 533-48, June 1999.

10. Ashish Mehra, Anees Shaikh, Tarek Abdelzaher, Zhiqun Wang, and Kang G. Shin, "Realizing Services for Guaranteed-QoS Communication on a Microkernel Operating System," *Proc. 19th IEEE Real-Time Systems Symposium*, Madrid, Spain, pp. 460–469, Dec. 1998.

11. H. Zou and F. Jahanian, "Optimization of a Real-Time Primary-Backup Replication Service," *Proceedings of the 17th Symposium on Reliable Distributed Systems (SRDS98)*, West Lafayette, pp. 177-85, Oct. 1998.

12. Tarek Abdelzaher and Kang G. Shin, "End-host architecture for QoS-adaptive communication," *IEEE RTAS98*, pp. 121–130, June 1998.

13. H. Zou and F. Jahanian, "Real-Time Primary-Backup Replication with Temporal Consistency Guarantees," *Proceedings of International Conf. on Distributed Computing Systems*, Amsterdam, The Netherlands, May 1998.

14. Atri Indiresan, Ashish Mehra, and Kang G. Shin, "The END: A network adapter design tool," *INFOCOM98*, April 1998.

15. S. Dawson, F. Jahanian, and T. Mitton, "Experiments on Six Commercial TCP Implementations Using a Software Fault Injection Tool," *Software Practice and Experience*, vol. 27, no. 12, pp. 1385-1410, December 1997.

16. T. Abdelzaher, S. Dawson, W.-C. Feng, F. Jahanian, S. Johnson, A. Mehra, T. Mitton, A. Shaikh, K. Shin, Z. Wang, and H. Zou, "ARMADA Middleware Suite," *IEEE Workshop on Middleware for Distributed Real-Time Systems and Services*, San Francisco, December 2, 1997.

17. A. Mehra, J. Rexford, H. Ang, and F. Jahanian, "Design and Evaluation of a Window-Consistent Replication Service," *IEEE Transactions on Computer*, vol. 46, no. 9, September 1997.

18. A. Mehra, A. Indiresan, and K. G. Shin, "Structuring communication software for quality-of-service guarantees," *Proc. 17th IEEE Real-Time Systems Symp.*, pp. 155–164, December 1996.

19. T. Abdelzaher, A. Shaikh, F. Jahanian, and K. Shin, "RTCAST: Lightweight Multicast for Real-Time Process Groups," *IEEE Real-Time Technology and Applications Symposium*, Best Student Paper Award, Brookline, Mass, pp. 250-259, June 1996.

20. S. Dawson, F. Jahanian, T. Mitton, and T.-L. Tung, "Testing Distributed Systems via Protocol Fault Injection," *FTCS-26*, Sendai, Japan, pp. 404-414, June 1996.

3 Platform

The services developed in the context of the ARMADA project are to augment the essential capabilities of a real-time microkernel by introducing a composable collection of communication, fault-tolerance, and testing tools to provide an integrated framework for developing and executing real-time applications. Most of these tools are implemented as separate servers in user space that can be colocated for efficiency considerations within the microkernel. Note that the advantages of server colocation within the microkernel do not defeat the purpose of using microkernels in the first place. This is because colocated servers (i) can be developed *in user space* which greatly reduces their development and maintenance cost, and (ii) can be *selectively included* into the kernel in accordance with the application requirements; this is both more efficient and more sensitive to particular application needs. Below we describe the experimental testbed and implementation environment common to the aforementioned services. A detailed description of the implementation approach adopted for various services will be given in the context of each particular service.

3.1 General Service Implementation Approach

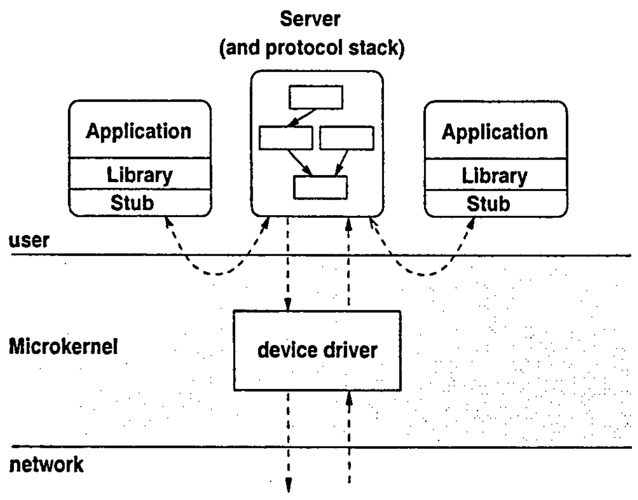
One common aspect of different middleware services in a distributed real-time system is their need to use inter-machine communication. All ARMADA services either include or are layered on top of a communication layer which provides the features required for correct operation of the service and its clients. For example, RTCAST implements communication protocols to perform multicast and integrate failure detection and handling into the communication subsystem. Similarly, the Real-Time Channels service implements its own signalling and data transfer protocols to reserve resources and transmit real-time data along a communication path. Since communication seemed to warrant particular attention in the context of this project, we developed a generic real-time communication subsystem architecture. The architecture can be viewed as a way of structuring the design of communication-oriented services for predictability, as opposed to being a service in itself. This architecture is described in detail in Section 4 and is illustrated by an example service: the Real-Time Channel. ARMADA communication services are generally layered on top of IP, or UDP/IP. We do not use TCP because its main focus is reliability as opposed to predictability and timeliness. Real-time communication protocols, on the other hand, should be sensitive to timeliness guarantees, perhaps overriding the reliability requirement. For example, in video conferencing and process control, occasional loss of individual data items is preferred to receiving reliable streams of stale data. To facilitate the development of communication oriented services, our communication subsystem is implemented using the x-kernel object-oriented networking framework originally developed at the University of Arizona [1], with extensions for controlled allocation of system resources [2]. The advantage of using x-kernel is the ease of composing protocol stacks. An x-kernel communication subsystem is implemented as a configurable graph of protocol objects. It allows easy reconfiguration of the protocol stack by adding or removing protocols. More details on the x-kernel can be found in [1].

Following the microkernel philosophy, our services are designed as user level servers. Clients of the service are separate processes that communicate with the server via the kernel using a user library. The library exports the desired middleware API. Communication-oriented services generally implement their own protocol stack that lies on top of the kernel-level communication driver. The x-kernel framework permits migration of the protocol stack into the operating system kernel. We use this feature to implement server colocation into the microkernel. Figure 3-a and 3-b illustrate the configurations of user level servers and colocated servers respectively. An example of server migration into the kernel is given in the context of the RTCAST service in Section 5. The RTCAST server was developed in user space (as in Figure 3-a), then reconfigured to be integrated the into the kernel (as in Figure 3-b). Whether the server runs in user space or is colocated in the microkernel client, processes use the same service API to communicate with it. If the service is colocated in the kernel, an extra context switch to/from a user level server process is saved. Automatically-generated stubs interface the user library (implementing the service API) to the microkernel or the server process. These stubs hide the details of the kernel's local communication mechanism from the programmer of the real-time service, thus making service code independent from specifics of the underlying microkernel.

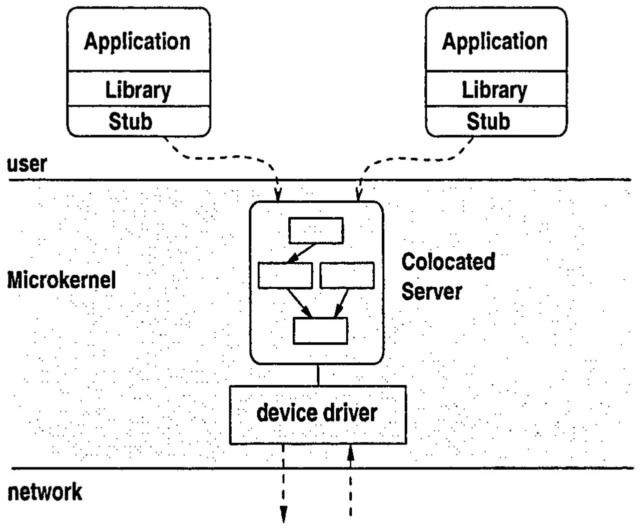
3.2 Testbed and Implementation Environment

In the following sections description is given of the implementation of each individual service. To provide a common context for that description, we outline here the specifics of the underlying implementation platform. Our testbed (see Figure 4) comprises several Pentium-based PCs (133 MHz) connected by a Cisco 2900 Ethernet switch (10/100 Mb/s), with each PC connected to the switch via 10 Mb/s Ethernet. We have chosen the MK 7.2 microkernel operating system from the Open Group (OG)[1] Research Institute to provide the essential underlying real-time support for our services. The MK microkernel is originally based on release 2.5 of the Mach operating system from CMU. While not a full-fledged real-time OS, MK 7.2 supports the x-kernel and includes several important features that facilitate provision of QoS guarantees. For example, MK 7.2 provides a unified framework for allocation and management of communication resources. This framework, known as *CORDS* (Communication Objects for Real-time Dependable Systems) [2], was found particularly useful for implementing real-time communication services. More detail on *CORDS* support and an example of using it is discussed in the context of presenting real-time channels in Section 4. Our implementation approach has been to utilize the functionality and facilities provided in OG's environment and augment it with our own support when necessary. The next

[1]Open Group is formerly known as the Open Software Foundation (OSF)



(a) User level server configuration



(b) Colocated server

Figure 3: Service implementation.

release of MK microkernel is expected to provide additional real-time support such as capacity reserves. This support is important for composability, in that it provides isolation of temporal behavior of individual services, so that predictability is maintained regardless of the deployed service mix.

4 ARMADA Real-Time Communication Architecture

In this section we present the overall architecture of the ARMADA real-time communication service. We first motivate and outline the goals and paradigm of real-time communication that we adopt. Subsequently, we highlight key aspects of the service architecture that we believe meets the stated goals.

4.1 Real-Time Communication Service: Goals and Paradigm

Our primary goal is to provide applications and middleware with a service that can be used to request and obtain (real-time) guaranteed-QoS unicast connections between two hosts. Such a service must satisfy three primary architectural requirements for guaranteed-QoS communication [3]: (i) maintenance of per-connection QoS guarantees, (ii) overload protection via per-connection traffic enforcement, and (iii) fairness to best-effort traffic. In order for the service to satisfy these requirements, an application must specify the worst-case traffic profile it expects to generate, and the end-to-end QoS it desires for different QoS parameters such as delay, bandwidth, and likelihood of packet loss. A secondary, but very important, goal is to design the service in a way that allows (a) constituent architectural components to be reused for other middleware services, and (b) flexibility to realize other QoS paradigms and service models. This is important because in our environment, multiple middleware services must coexist and interoperate; reusing architectural components, whenever possible, makes service integration relatively easier. Equally important, in order to realize generic components, one must decouple the service model from the component architecture. This facilitates the ability to extend the service to more relaxed QoS models such as probabilistic guarantees, and QoS negotiation and degradation.

To realize the real-time communication service, we adopt the service model of *real-time channels* [4, 5], a paradigm for guaranteed-QoS communication in packet-switched networks. Not only is this model similar to other proposals for guaranteed-QoS communication [6], we have developed significant insights in extending the

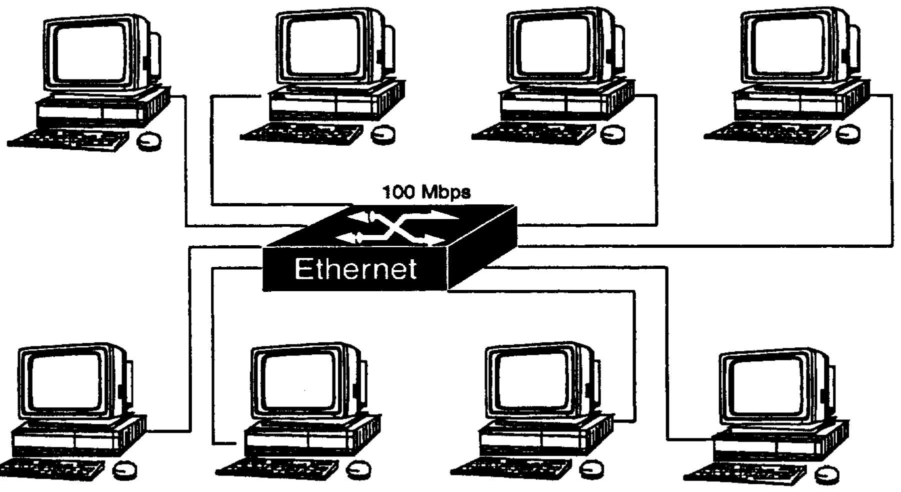


Figure 4: Experimental testbed.

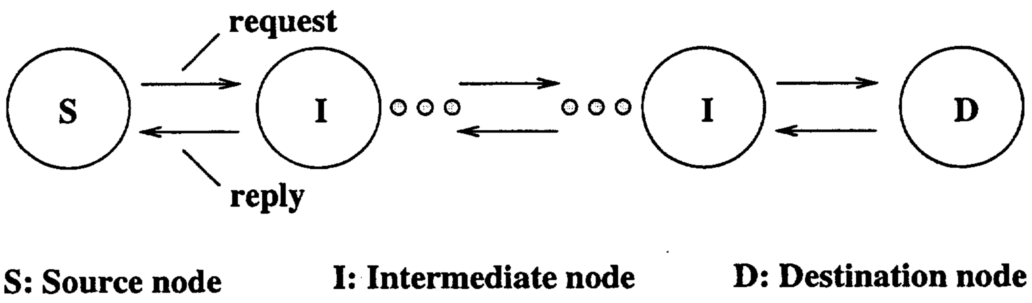


Figure 5: Signalling between two hosts.

model appropriately for practical use [7, 3]. A real-time channel is a simplex, fixed-route, virtual connection between a source and a destination host, with sequenced messages and associated performance guarantees on message delivery. Real-time communication via real-time channels is performed in three phases (see Figure 5). In the first phase, the source host S (sender) creates a channel to the destination host D (receiver) by specifying the channel's traffic parameters and QoS requirements. Signalling requests are sent from S to D via one or more intermediate (I) nodes; replies are delivered in the reverse direction from D to S. If successfully established, S can send messages on this channel to D; this constitutes the second phase. When the sender is done using the channel, it must close the channel (the third phase) so that resources allocated to this channel can be released.

Traffic and QoS Specification: Traffic generation on real-time channels is based on a *linear bounded arrival process* [8, 9] characterized by three parameters: maximum message size (M_{max} bytes), maximum message rate (R_{max} messages/second), and maximum burst size (B_{max} messages). The notion of *logical arrival time* is used to enforce a minimum separation $I_{min} = \frac{1}{R_{max}}$ between messages on the same real-time channel. This ensures that a channel does not use more resources than it reserved at the expense of other channels. The QoS on a real-time channel is specified as the desired deterministic, worst-case bound on the end-to-end delay experienced by a message. See [5] for more details.

Resource Management: Admission control for real-time channels is provided by Algorithm `D_order` [5], which uses fixed-priority scheduling for computing the worst-case delay experienced by a channel at a link. When a channel is to be established at a link, the worst-case response time for a message (when the message completes transmission on the link) on this channel is estimated based on non-preemptive fixed-priority scheduling of packet

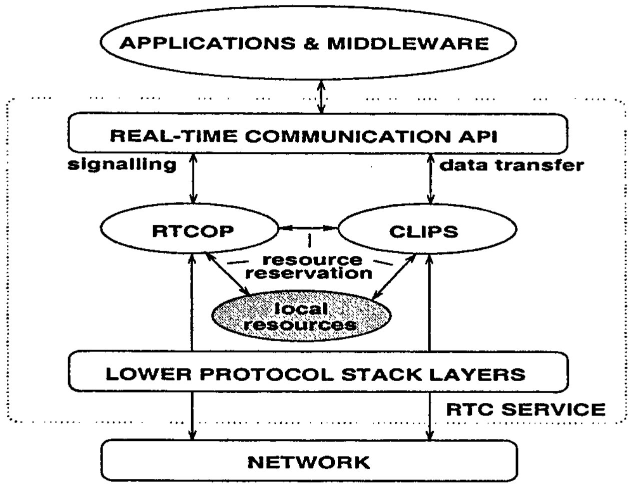


Figure 6: ARMADA services and software architecture for real-time communication.

transmissions. The total response time, which is the sum of the response times over all the links on the route of the channel, is checked against the maximum permissible message delay and the channel established only if the latter is greater. A local delay bound is derived from the worse-case response time and the specified end-to-end delay bound. Run-time link scheduling, on the other hand, is governed by a multi-class variation of the earliest-deadline-first (EDF) policy.

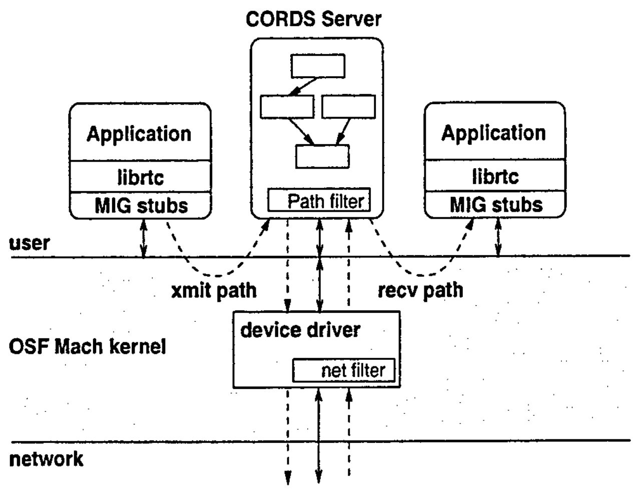
QoS-sensitive Communication Subsystem: We have proposed and evaluated a QoS-sensitive communication subsystem architecture realizing real-time channels [3]. The architecture delivers contracted QoS to individual channels, provides isolation between connections, manages memory and CPU resources in accordance with priorities derived from QoS specification, performs admission control to protect established QoS guarantees, and uses policing to isolate misbehaving traffic. An instantiation of this architecture is described in Section 4.2.

4.2 Service Architecture

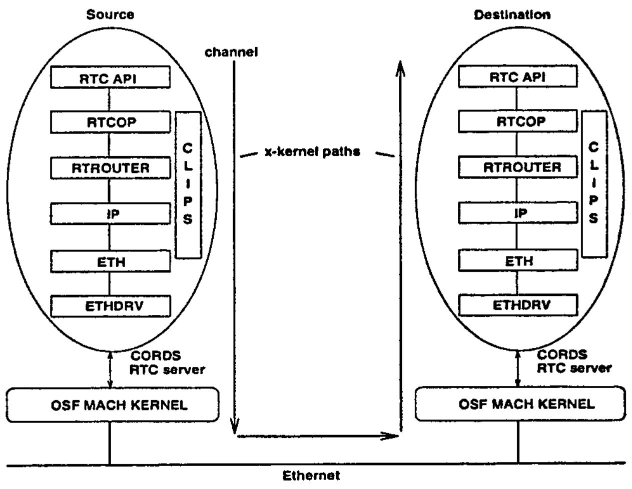
Figure 6 illustrates the software architecture of our guaranteed-QoS service at hosts and intermediate nodes. The core functionality of the service is realized via three distinct components that interact to provide guaranteed-QoS communication: real-time communication API (RTC API), RTCOP, and CLIPS. While applications use the service via the RTC API, end-to-end signalling is coordinated by RTCOP and run-time management of resources for data transfer performed by CLIPS. As mentioned above, the run-time resource management in the service architecture is based in large part on the architecture proposed in [3], with significant enhancements to accommodate the specific requirements of the ARMADA project and its implementation environment. Below we give a brief overview of the three components of our service architecture. Details on the internals of the service components and their interaction are provided in [10].

Invocation via RTC API: Applications establish and teardown guaranteed-QoS connections, and perform data transfer on these connections, by invoking routines exported by the RTC API. The API can be viewed as comprising two parts. The top half interfaces to applications and is responsible for validating application requests and creating internal state. The bottom half interfaces to RTCOP for signalling, i.e., connection setup and teardown, and to CLIPS for QoS-sensitive data transfer. The design of the API has been significantly influenced by the structure of the sockets API in BSD Unix and its variants.

Signalling via RTCOP: End-to-end *signalling and resource reservation* is performed by RTCOP to establish and teardown guaranteed-QoS connections across the communicating hosts, possibly via multiple network nodes. RTCOP provides reliable datagram semantics for signalling requests and replies between nodes, and performs



(a) CORDS server configuration



(b) Protocol stack

Figure 7: Service implementation as CORDS server.

consistent channel state management at each node. It interfaces to the admission control module (that keeps track of available communication resources) to admit new connection requests, and establish appropriate connection state to store connection-specific information. Similarly, it interfaces to the routing module to select a (unicast) path on which to perform the end-to-end signalling. As shown in Figure 6, it interfaces to CLIPS and directly to local resources for resource reservation functions.

Data transfer via CLIPS: CLIPS facilitates *network data transport* on established real-time channels, providing services for management of CPU and link resources during data transfer in order to maintain QoS guarantees. For example, it provides services for allocation of protocol processing resources and fragmentation of application data (messages) into smaller units (packets) for network transmission. In addition to managing CPU resources for protocol processing, CLIPS also performs *link access scheduling* to manage link bandwidth such that all active connections receive their promised QoS. This involves abstracting the link in terms of transmission delay and bandwidth, and scheduling all outgoing packets for network access. The minimum requirement for provision of QoS guarantees is that packet transmission time on the link be bounded and predictable. CLIPS also performs *traffic enforcement* on a per-channel basis, forcing an application to conform to the channel's traffic specification and provide overload protection between established connections. The design of CLIPS has been motivated by the requirements of the different ARMADA real-time communication and middleware services.

4.3 Prototype Implementation

We have developed a prototype implementation of the service architecture described in the preceeding section. Below we briefly describe the implementation environment, followed by a description of the implementation approach adopted. Additional details on the implementation, parameterization, and evaluation of the service are provided in [10].

4.3.1 Implementation environment

Figure 7 shows the software configuration we have realized for our service implementation using the CORDS [2] framework. As shown in Figure 7(a), applications link with a library, `librtc`, which implements RTC API and interfaces to the RTC service on behalf of the application. `librtc` is in turn realized on top of MIG (Mach Interface Generator) interface stubs which provide the necessary support for Mach IPC. Each library routine translates to an IPC call to the top-level anchor protocol in the CORDS server (Figure 7(b)), which interacts with the rest of the protocol stack on behalf of the application.

CORDS: CORDS is based on the x-kernel object-oriented networking framework originally developed at the University of Arizona [1], with some significant extensions for controlled allocation of system resources. The primary advantage of using CORDS for our prototype implementation is its support for communication resource allocation. It provides a unified mechanism for allocating memory and CPU thread resources to individual communication protocols on machines along a connection path from source to destination. Such support facilitates resource management to maintain *per-connection* QoS guarantees which plays a significant role in the realization of our service architecture on this platform. Another advantage of using CORDS is the ease of composing protocol stacks in the x-kernel networking framework.

The Path Framework: While preserving the structure and functionality of the original x-kernel, CORDS adds two abstractions, *paths* and *memory allocators*, to provide an interface for path-specific reservation/allocation of system resources such as message buffers, dynamically allocated memory, input packet buffers, and threads that shepherd messages up a protocol stack [2]. Paths, coupled with allocators, provide a capability for reserving and allocating resources at any protocol stack layer on behalf of a particular class of messages. With packet demultiplexing at the lowest level (device driver), it is possible to isolate packets on different paths from each other. Incoming packets are stored in buffers explicitly tied to the appropriate path and serviced by threads previously allocated to that path. Moreover, threads reserved for a path may be assigned one of several scheduling policies and priority levels.

Paths greatly facilitate realization of a connection-oriented protocol stack, since the knowledge of paths is available at any layer in the protocol stack. However, the CORDS framework envisions a relatively static use of paths, with a single path for best-effort traffic and a few paths for different classes of traffic. That is, there are never more than perhaps ten active paths, all of these long-lived and preconfigured. Accordingly, the CORDS path library provides no support for path teardown or resource reclamation, operations which are necessary in a dynamic environment in which paths are created and destroyed frequently. To realize a one-to-one mapping between real-time channels and paths, we have extended the path framework with support for path destruction and reclamation of resources associated with a path.

4.3.2 Service configuration

While the CORDS framework can be used at user-level as well in the kernel, we have developed the prototype implementation as a user-level CORDS server. There are several reasons for this choice. The first and most obvious is the ease of development and debugging, resulting in a shorter development cycle. While this is true in general, it is particularly critical when handling timing-related software and the associated bugs. The second reason is the infeasibility of the other alternative (to pure in-kernel development) for our implementation. CORDS allows an x-kernel protocol graph to span address space (and protection) boundaries via *proxies* [2]. Thus, our implementation could have spanned user and kernel space, with portions of our implementation developed in user-space and moved into the kernel after debugging and testing. However, this was not feasible for two reasons: (i) strong inter-dependencies between the service components, and (ii) interaction of CLIPS with multiple layers of the protocol stack. It was important, therefore, to keep all service components in one address space, i.e., in the CORDS server.

Although this design is prone to high overhead in terms of additional IPC context switches, the server is designed such that it may be easily moved into the kernel in subsequent versions. The bottom layer of the CORDS protocol stack in the server interfaces with the kernel device driver via a Mach device port. The link scheduler is implemented here, as close as possible to the network without being in the kernel. The kernel driver has mechanisms to route incoming packets to the corresponding communication server (via a net filter). The

CORDS server, in turn, employs a path filter to route the packets to the right pool of packet buffers and shepherd threads. The net filter and path filter are shown in Figure 7(a) representing this mechanism.

4.3.3 Protocol stack

The CORDS-based protocol stack for real-time communication is shown in Figure 7(b). Protocols comprising this stack include the RTC API anchor, CLIPS, RTCOP, RTROUTER, IP, ETH, and ETHDRV; these are briefly described below.

RTC API Anchor: The anchor protocol for RTC API is the top-level protocol configured directly above RTCOP. It interfaces on the top with the applications via the MIG stubs, translating application requests to specific invocations of operations on RTCOP (for signaling) or CLIPS (for data transfer). It maintains several mappings for real-time channels, such as the association between Mach data and signalling ports, RTCOP ports, and *x*-kernel sessions. It also keeps track of appropriate client state for client death cleanup.

CLIPS: CLIPS spans several layers of the protocol stack, exporting an interface that is used by RTC API, RTCOP, and ETHDRV. CLIPS provides a generic mechanism for prioritized, time-bounded message processing and communication at end hosts. Specifically, it implements support for associating communication endpoints of a protocol stack with *clips*, an abstraction to which CLIPS allocates resources. It also implements QoS-sensitive CPU and link bandwidth allocation, and provides message queues for traffic originating from the RTC API.

RTCOP: RTCOP is realized as an *x*-kernel protocol residing above a two-part network layer composed of RTROUTER and IP. It exports an interface to RTC API for specification of channel establishment and teardown requests and replies, and selection of logical ports for channel endpoints. RTCOP also utilizes four other interfaces during signalling: an (internal) interface to admission control, an interface to RTROUTER, an interface to CLIPS, and an interface to local path-specific system resources such as packet queues, buffers, and thread pools.

RTROUTER: Real-time channels currently use the default IP routing. However, to keep the routing interface independent of IP, we provide RTROUTER as a go-between protocol. RTROUTER is intended to allow RTCOP to work with more sophisticated routing protocols that support QoS- or policy-based routing. In the current implementation, RTROUTER performs the following functions: (i) specifies the connectivity of real-time channel nodes with its pre-configured topology tables, (ii) provides a logical *RTHost* (real-time host) addressing mechanism for real-time channels, and (iii) handles forwarding of RTCOP packets between source and destination RTHosts.

IP, ETH, ETHDRV: The IP, ETH, and ETHDRV protocols are standard implementations distributed with the CORDS framework. ETH is a generic hardware-independent protocol that provides an interface between higher level protocols and the actual Ethernet driver. ETHDRV is specific to a particular user-level implementation of the CORDS server. It is an out-of-kernel device driver that interacts with the network device driver in the Mach kernel through system calls to a Mach device control port. We have made modifications to the ETHDRV protocol to realize the link scheduler and extensions for resource reclamation on paths.

5 RTCAST Group Communication Services

The previous section introduced the architecture of the ARMADA real-time communication service. The second thrust of the project has focused on a collection of modular and composable middleware services for constructing embedded applications. The ARMADA middleware can be divided into two relatively independent suites of services:

- RTCAST group communication services, and

- RTPB real-time primary-back replication service.

This section presents the RTCAST suite of group communication and fault-tolerance services. Section 6 describes the RTPB (real-time primary-backup) replication service.

5.1 RTCAST Protocols

Process groups are a widely-studied paradigm for designing dependable distributed systems in both asynchronous [11, 12, 13, 14] and synchronous [15, 16, 17] environments. In this approach, a distributed system is structured as a group of cooperating processes which provide service to the application. A process group may be used, for example, to provide active replication of system state or to rapidly disseminate information from an application to a collection of processes. Two key primitives for supporting process groups in a distributed environment are *fault-tolerant multicast communication* and *group membership*. Coordination of a process group must address several subtle issues including delivering messages to the group in a reliable fashion, maintaining consistent views of group membership, and detecting and handling process or communication failures. If multicast messages are atomic and globally ordered, process group members can be kept consistent when process state is determined by initial state and the sequence of received messages.

We developed RTCAST to provide a lightweight fault-tolerant multicast and membership service for real-time process groups which exchange periodic and/or aperiodic messages. The RTCAST group communication does not require acknowledgments for every message, and message delivery is immediate without needing additional "rounds" of message transmissions. RTCAST is designed to support hard real-time guarantees without requiring a static schedule to be computed *a priori* for application tasks and messages. Instead, an on-line schedulability analysis component performs admission control on multicast messages. We envision the proposed multicast and membership protocols as part of a larger suite of middleware group communication services that form a composable architecture for the development of embedded real-time applications. As illustrated in Figure 8, the RTCAST suite of services include a timed atomic multicast, a group membership service and an admission control service. The first two are tightly coupled and thus are considered a single service. Clock synchronization is typically required for real-time protocols and is enforced by the clock synchronization service. To support portability, a *virtual network interface* layer exports a uniform network abstraction. Ideally, this interface would transparently handle different network topologies, each having different connectivity and timing or bandwidth characteristics exporting a generic network abstraction to upper layers. The network is assumed to support unicast datagram service. Finally, the top layer provides an application programming interface for real-time process group.

RTCAST supports bounded-time message transport, atomicity, and order for multicasts within a group of communicating processes in the presence of processor crashes and communication failures. It also supports prioritized message delivery, where the application can assign messages to one of several priority classes. Messages from higher priority classes are served before any waiting messages of lower priority classes, resulting in improved performance for application-critical messages. Atomicity and message ordering are still enforced on a per-priority basis. It guarantees agreement on membership among the communicating processors, and ensures that membership changes (e.g., resulting from processor joins or departures) are atomic and ordered with respect to multicast messages. RTCAST proceeds as senders in a logical ring take turns in multicasting messages over the network. A processor's turn comes when the logical token arrives, or when it times out waiting for it. After its last message, each sender multicasts a heartbeat that is used for crash detection. The heartbeat received from an immediate predecessor also serves as the logical token. Destinations detect missed messages using sequence numbers and when a processor detects a receive omission, it crashes. Each processor, when its turn comes, checks for missing heartbeats and eliminates the crashed members, if any, from group membership by multicasting a membership change message.

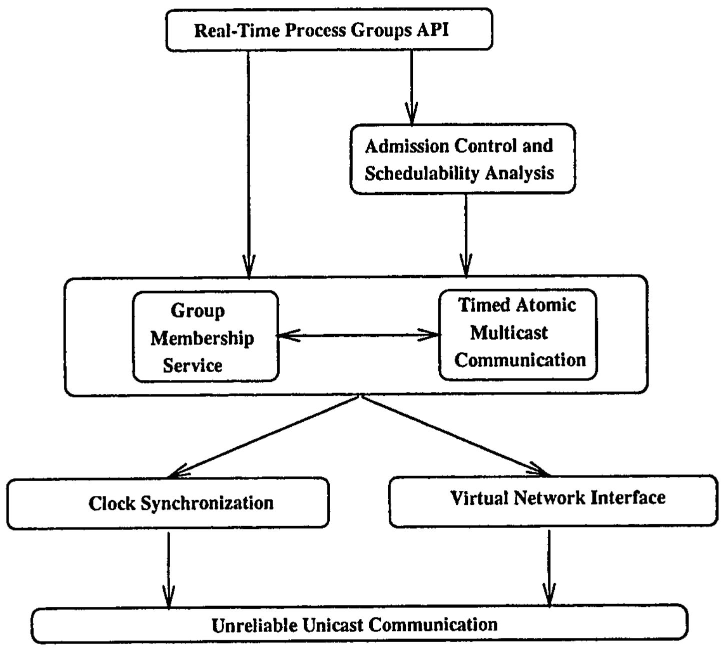

Real-Time Process Groups API
Admission Control and Schedulability Analysis
Group Membership Service
Timed Atomic Multicast Communication
Clock Synchronization
Virtual Network Interface
Unreliable Unicast Communication

Figure 8: Software architecture for the RTCAST middleware services.

In a token ring, sent messages have a natural order defined by token rotation. We reconstruct message order at the receivers using a protocol layer below RTCAST which detects out-of-order arrival of messages and swaps them, thus forwarding them to RTCAST in correct order. RTCAST ensures that "correct" members can reach agreement on replicated state by formulating the problem as one of group membership. Since the state of a process is determined by the sequence of messages it receives, a processor that detects a message receive omission takes itself out of the group, thus maintaining agreement among the remaining ones. In a real-time system one may argue that processes waiting for a message that does not arrive will miss their deadlines anyway, so it is acceptable to eliminate the processor(s) which suffered receive omissions.[2] A distinctive feature of RTCAST is that processors which did not omit any messages can deliver messages as soon as they arrive without compromising protocol semantics. Thus, for example, if a reliable multicast is used to disseminate a critical message to a replicated server, and if one of the replicas suffers a receive omission, RTCAST will eliminate that replica from the group, while delivering the message to the remaining replicas immediately. This is in contrast to delaying delivery of the message until *all* replicas have received it. The approach is motivated by the observation that in a real-time system it may be better to sacrifice *one* replica in the group than delay message delivery potentially causing *all* replicas to miss a hard timing constraint. Finally, membership changes are communicated exclusively by *membership change messages* using our multicast mechanism. Since message multicast is atomic and ordered, so are the membership changes. This guarantees agreement on membership view.

From an architectural standpoint, RTCAST operation is triggered by two different event types, namely message reception, and token reception (or timeout). It is therefore logically structured as two event handlers, one for each event type. The **message reception handler** (Figure 9) detects receive omissions if any, delivers messages in order to the application, and services protocol control messages. The **token handler** (Figure 10) is invoked when the token is received or when the token timeout expires. It detects processor crashes and sends membership change notifications, if any, as well as lets client processes send out their messages during the processors finite token hold time.

[2]A lower communication layer may support a bounded number of retransmissions.

```
msg_reception_handler()
1 if state = RUNNING
2  if more msgs from same member
3   if missed msgs → CRASH else
4    deliver msg
5  else if msg from different member
6   if missed msgs → CRASH else
7    check for missed msgs from processors
      between current and last senders
8    if no missing msgs
9     deliver current msg
10   else CRASH
11 else if join msg from non-member
12  handle join request
13if state = JOINING AND
   msg is a valid join_ack
14 if need more join_acks
15  wait for additional join_acks
16 else state = RUNNING
end
```

Figure 9: Message reception handler

```
token_handler()
1 if (state = RUNNING)
2  for each processor p in
    current membership view
3   if no heartbeat seen from all predecessors
     up to and including p
4    remove p from group view
5    multicast new group view
6  send out all queued messages
7  mark the last msg
8  send out heartbeat msg
9 if (state = JOINING)
10 send out join msg
end
```

Figure 10: Token handler

5.2 RTCAST Design and Implementation

This section describes some of the major issues in the design and implementation of RTCAST; our representative group communication service. A thorough performance evaluation of the service is presented in [18] and [19].

The RTCAST application was implemented and tested over a local Ethernet. Ethernet is normally unsuitable for real-time applications due to packet collisions and the subsequent retransmissions that make it impossible to impose deterministic bounds on communication delay. However, since we use a *private* Ethernet (i.e. the RTCAST protocol has exclusive access to the medium), only one machine can send messages at any given time (namely, the token holder). This prevents collisions and guarantees that the Ethernet driver always succeeds in transmitting each packet on the first attempt, making message communication delays deterministic. The admission control service described previously can take advantage of this predictability to make real-time guarantees on messages sent with RTCAST, and to schedule messages for transmission so that they can be delivered by their deadlines.

5.2.1 Protocol Stack Design

The RTCAST protocol was designed to be modular, so that individual services could be added, changed, or removed without affecting the rest of the protocol. Each service is designed as a separate protocol layer within the *x*-kernel [1] protocol framework. The *x*-kernel is an ideal choice for implementing the RTCAST middleware services because application requirements can be easily met by simply reconfiguring the protocol stack to add or remove services as necessary. The RTCAST implementation uses the following protocol layers:

Admission Control The Admission Control and Schedulability Analysis (ACSA) layer performs message scheduling and real-time delivery guarantees based on the predictable group delivery service provided by the RTCAST protocol itself. If deadline-based delivery guarantees are not needed, this layer can be omitted from the protocol stack to reduce overhead.

RTCAST The RTCAST protocol layer encompasses the membership, logical token ring, and atomic ordering services described in section 5.

Multicast Transport This protocol implements an unreliable multicast abstraction that is independent of the underlying network. RTCAST uses the multicast transport layer to send messages to the group without having to worry about whether the physical medium provides unicast, broadcast, or true multicast support. The details of how the messages are actually sent over the network are hidden from higher layers by the multicast transport protocol, so it is the only layer that must be modified when RTCAST is run on different types of networks.

IP,ETH,ETHDRV These are the same as described in Section 4.3.3.

Figure 11 shows the full protocol stack as it is implemented on our platform.

5.2.2 Integration Into the Mach Kernel

As figure 11 shows, the protocol stack representing the core of the service was migrated into the Mach kernel. While actual RTCAST development took place in user space to facilitate debugging, its final colocation within the Mach kernel has several performance advantages. First, as with any group communication protocol, there can be a high amount of CPU overhead to maintain the group state and enforce message semantics. By running in the kernel, the RTCAST protocol can run at the highest priority and minimize communication latency due to processing time. Second, in the current implementation of MK 7.2 there is no operating system support

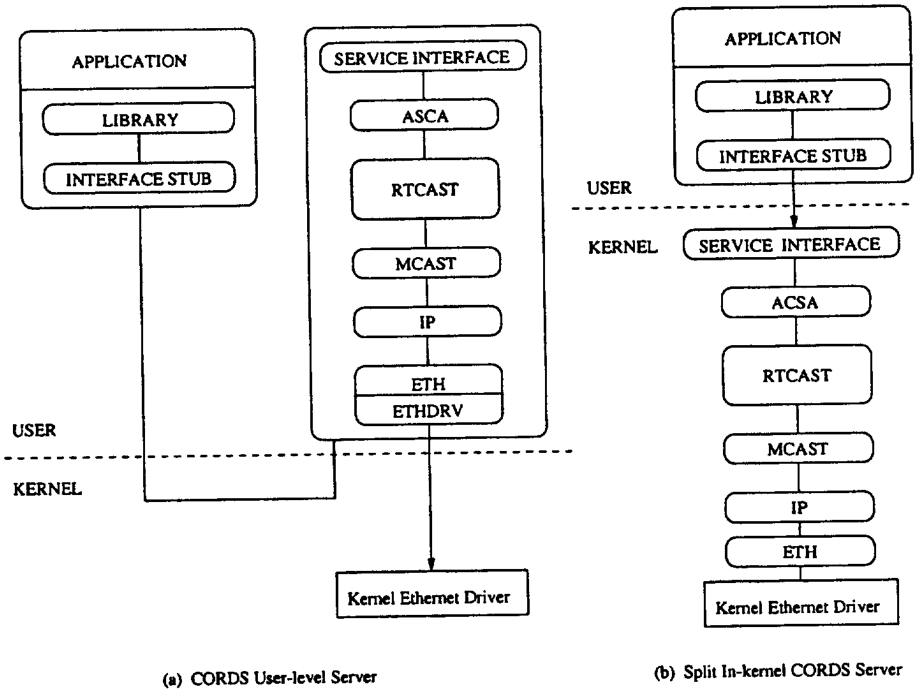


Figure 11: RTCAST protocol stack as implemented

for real-time scheduling or capacity reserve. Experience shows that processes running at the user level can be starved for CPU time for periods of up to a few seconds, which would be disastrous for RTCAST's predictable communication. By running in the kernel, protocol threads do not get starved significantly and are scheduled in a much more predictable manner by the operating system. Finally, there is a problem with the MK 7.2 implementation of the *x*-kernel, such that threads which are shepherding messages up the protocol stack can be queued to run in a different order than the messages arrive from the network. This results in out-of-order messages that must be buffered and re-ordered to maintain the total ordering guarantees provided by the protocol. Having to buffer and reorder messages also delays crash detection, since there is no way of knowing if a missing message is queued somewhere in the protocol stack or if the sender suffered a failure. By running the protocol in the kernel, message threads are interrupt driven and run immediately after arriving from the network, so the message reordering problem does not occur.

5.2.3 Other RTCAST Implementations

RTCAST has also been ported to Solaris, Linux, and Windows NT. The Linux version has been integrated with the Linux/RK resource kernel from CMU, which provides a CPU reservation service that we utilize to ensure RTCAST's predictability and timeliness. This allows us to provide much stronger real-time guarantees than can currently be supported by any other platform. This implementation is described in [20], along with a performance analysis of its timeliness and predictability.

5.2.4 Prioritized Group Communication

RTCAST supports prioritized group communication using a new abstraction we have developed, the *priority queue*. This abstraction enables the application to assign priorities to messages, and the communication protocol will give preference to serving higher priority messages over lower priority ones. Priorities are also considered in the real-time scheduling of message deliveries. If the protocol can guarantee the delivery of a higher-priority message by violating the deadline of an already guaranteed lower-priority message, it will do so to better meet the application's stated priorities regarding communication. Even in non-real-time systems, this abstraction can improve the predictability of communication performance, since messages of a given priority class tend to have similar latencies compared to messages from different priority classes.

5.2.5 Interfacing to Applications

Another important focus in developing our group communication middleware was designing a robust API that would allow application developers to take advantage of our services quickly and easily. RTCAST API includes (i) bandwidth reservation calls, (ii) process group membership manipulation functions, (iii) best effort communication primitives and (iv) reliable real-time unicast and multicast. Bandwidth reservation is used on hosts to ensure that a multicast connection has dedicated CPU capacity and network bandwidth (i.e. a minimum token hold time). This may be used to provide schedulability guarantees at a higher level. The membership manipulation functions allow processes to join and leave the multicast group, query current group membership, create groups, etc. There are two types of group communication: real-time communication that guarantees end-to-end response time, and best effort which does not. The advantages of using a best effort connection is that it is optimized for throughput as opposed to meeting individual message deadlines. Thus, the service protocol stack is faster on the average (e.g., no per-message admission control), but the variance in queuing delays is higher.

We collaborated with a group of researchers at the Honeywell Technology Center to implement a subset of the fault-tolerant real-time distributed application described in Section 1 using the RTCAST protocol. Using the insights gained from this motivating application, we were able to refine the API to provide the required of functionality while maintaining a simple interface that is easy to program. Based on our experience of the application's use of the protocol, we also designed a higher-level service library that can be linked with the application, and which uses the RTCAST API. It is concerned with resource management in a fault-tolerant system and with providing higher-level abstractions of the protocol communication primitives. The service library provides for logical processing nodes and resource pools that transparently utilize RTCAST group communication services. These abstractions provide a convenient way for application developers to reason about and structure their redundancy management and failure handling policies while RTCAST does the actual work of maintaining replica consistency. The APIs for both the service library and the RTCAST protocol are available at http://www.eecs.umich.edu/RTCL/armada/rtcast/api.html.

5.3 Other Group Communication Services

In addition to the RTCAST group communication protocol, we have developed a framework for composing multiple process groups together to build complex, large-scale systems. This framework is based on a new abstraction, the *inter-group router*, which enables two process groups to exchange messages without requiring the group communication middleware for either group to communicate with the other group directly. This reduces shared state in the system, improving scalability and simplifying design and failure recovery.

Using inter-group routers, groups can be composed together into larger systems while still maintaining well-defined end-to-end delivery semantics such as causal or total order. We have identified a number of basic composition topologies which can be used by the system designer to achieve a given end-to-end delivery semantic. These topologies can then be combined as needed to form larger systems.

This method for building distributed systems takes advantage of the natural structure of such systems, which are typically composed of many functional components operating in parallel to complete a particular task. Normally, in order to take advantage of group communication services, processes for all of these components would have to be placed in a single process group. This greatly magnifies the amount of shared state that must be maintained by the system, as well as its communication overhead. In addition, failures or other events occuring in one component can affect the entire system since all processes belong to the same group.

Using our architecture, these components can be composed independently in separate process groups, simplifying system design. Failures or other events in one component will now not affect other components, since they belong to different groups. Furthermore, overhead is reduced since messages exchanged entirely within a single component are not seen or processed by processes in other components. This allows the system to make use of group communication services without having to manage the extra complexity that comes from having large numbers of unrelated processes residing in the same group.

One of the advantages of the inter-group router abstraction is that it is independent of any particular group communication protocol. It operates between the group communication middleware and the application, intercepting and forwarding messages as necessary, so the underlying group communication middleware is not even aware of its existing. To the group communication protocol, the inter-group routers look like part of the application, and the protocol is not even aware of the existence of other groups. This allows any group communication protocol to utilize the inter-group routers without modification, and without having to do extra work or maintain extra shared state about other groups in the system.

For a more detailed description of our group composition architecture, please see [21].

6 Real-Time Primary-backup (RTPB) Replication Service

While the previous section introduced a middleware service for active replication, in this section we present the overall architecture of the ARMADA real-time primary-backup replication service. We first give an introduction to the RTPB system, then describe the service framework. Finally we discuss implementation of the service that we believe meets the objectives.

6.1 Introduction to RTPB

Keeping large amounts of application state consistent in a distributed system, as in the state machine approach, may involve a significant overhead. Many real-time applications, however, can tolerate minor inconsistencies in replicated state. Thus, to reduce redundancy management overhead, our primary-backup replication exploits application data semantics by allowing the backup to maintain a less current copy of the data that resides on the primary. The application may have distinct tolerances for the staleness of different data objects. With sufficiently recent data, the backup can safely supplant a failed primary; the backup can then reconstruct a consistent system state by extrapolating from previous values and new sensor readings. However, the system must ensure that the distance between the primary and the backup data is bounded within a predefined time window. Data objects may have distinct tolerances in how far the backup can lag behind before the object state becomes stale. The challenge is to bound the distance between the primary and the backup such that consistency is not compromised, while minimizing the overhead in exchanging messages between the primary and its backup.

6.2 Service Framework

A very important issue in designing a replication service is its consistency semantics. One category of consistency semantics that is particular relevant to the primary-backup replication in a real-time environment is *temporal consistency*, which is the consistency view seen from the perspective of the time continuum. Two types of temporal consistency are often needed to ensure proper operation of a primary-backup replicated real-time data services system. One is the *external temporal consistency* between an object of the external world and its image on the servers, the other is the *inter-object temporal consistency* between different objects or events.

A primary-backup system is said to satisfy the external temporal consistency for an object i if the timestamp of i at the server is no later than a predetermined time from its timestamp at the client (the real data). In other words, in order to provide meaningful and correct service, the state of the primary server must closely reflect that of the actual world. This consistency is also needed at the backup if the backup were to successfully replace the primary when the primary fails. The consistency restriction placed on the backup may not be as tight as that on the primary but must be within a tolerable range for the intended applications.

The inter-object temporal consistency is maintained if for any object pair, their temporal constraint δ_{ij} (which is the temporal distance of any two neighboring updates for object i, and j, respectively) is observed at both primary and backup.

Although the usefulness or practical application of the *external temporal consistency* concept is easy to see, the same is not true for *inter-object temporal consistency.* To illustrate the notion of inter-object temporal consistency, considering an airplane during taking off. There is a time bound between accelerating the plane and the lifting of the plane into air because the runway is of limited length and the airplane can not keep accelerating on the runway indefinitely without lifting off. In our primary-backup replicated real-time data service, the inter-object temporal consistency constraint between an object pair placed on the backup can be different from that placed on the primary.

6.3 RTPB Implementation

A temporal consistency model for the Real-time Primary-backup (RTPB) replication service has been developed [22] and a practical version of the system that implements the models has been built. Following our composability model, the RTPB service is implemented as an independent user-level x-kernel based server on our MK 7.2 based platform. Our system includes a primary server and a backup server. A client application resides in the same machine as the primary. The client continuously senses the environment and periodically sends updates to the primary. The client accesses the server using a library that utilizes the Mach IPC-based interface. The primary is responsible for backing up the data on the backup site and limiting the inconsistency of the data between the two sites within some required window. The following assumptions are made in the implementation:

- Link failures are handled using physical redundancy such that network partitions are avoided.
- An upper bound exists on the communication delay between the primary and the backup. Missed message deadlines are treated as communication performance failures.
- Servers are assumed to suffer crash failures only.

Figure 12 shows our system architecture and the x-kernel protocol stack for the replication server. The bottom five layers (RTPB to ETHDRV) make up the x-kernel protocol stack. At the top level of the stack is our real-time primary-backup (RTPB) protocol. It serves as an anchor protocol in the x-kernel protocol stack. From above, it provides an interface to the x-kernel based server. From below, it connects with the rest of the protocol stack through the x-kernel uniform protocol interface. The underlying transport protocol is UDP. Since UDP does not provide reliable delivery of messages, we need to use explicit acknowledgments when necessary.

The top two layers are the primary-backup hosts and client applications. The primary host interacts with the backup host through the underlying RTPB protocol. There are two identical versions of the client application residing on the primary and backup hosts respectively. Normally, only the client version on the primary is running. But when the backup takes over in case of primary failure, it also activates the backup client version and bring it up to the most recent state.

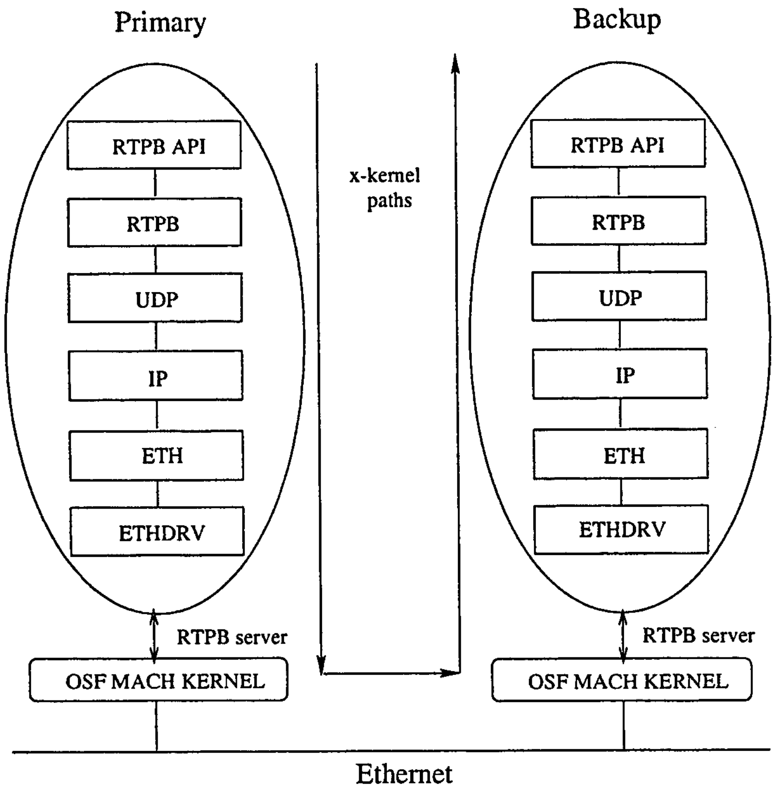


Figure 12: RTPB architecture and server protocol stack

The client application interacts with the RTPB system through the Mach API interface we developed for the system. The interface enables the client to create, destroy, manipulate and query reliable objects (i.e., those backed-up by our server). Specifically, *rtpb_create, rtpb_destroy* creates objects on and destroys objects from the RTPB system; *rtpb_register* register objects with the system; *rtpb_update, rtpb_query* update and query objects; finally *rtpb_list* return a list of objects that are already registered with the RTPB system. Further detail on admission control, update scheduling, failure detection and recovery appears in a recent report [22].

7 Evaluation Tools

The third thrust of the ARMADA project is to provide tools for validating and evaluating the timeliness and fault tolerance capabilities of the target system. Two tools have been developed to date: ORCHESTRA, a message-level fault injection tool for validation and evaluation of communication and middleware protocols, and COGENT, a network traffic workload generator. The following two subsections describe the two tools briefly.

7.1 Orchestra

Ensuring that a distributed system meets its prescribed specification is a growing challenge that confronts software developers and system engineers. Meeting this challenge is particularly important for applications with strict dependability and timeliness constraints. ORCHESTRA is a fault injection environment which can be used to perform fault injection on communication protocols and distributed applications. ORCHESTRA is based on a simple yet powerful framework, called *script-driven probing and fault injection*. The emphasis of this approach is on experimental techniques intended to identify specific "problems" in a protocol or its implementation rather than the evaluation of system dependability through statistical metrics such as fault coverage (e.g. [23]). Hence, the focus is on developing fault injection techniques that can be employed in studying three aspects of a target protocol: i) detecting design or implementation errors, ii) identifying violations of protocol specifications, and iii) obtaining insights into the design decisions made by the implementors.

In the ORCHESTRA approach, a fault injection layer is inserted into the communication protocol stack below the protocol to be tested. As messages are exchanged between protocol participants, they pass through the fault injection layer on their path to/from the network. Each time a message is sent, ORCHESTRA runs a script called the *send filter* on the message. In the same manner, the *receive filter* is invoked on each message that is received from the network destined for the target protocol. The scripts perform three types of operations on messages:

- **Message filtering:** for intercepting and examining a message.
- **Message manipulation:** for dropping, delaying, reordering, duplicating, or modifying a message.
- **Message injection:** for probing a participant by introducing a new message into the system.

The ORCHESTRA toolset on the MK 7.2 platform is based on a portable fault injection core, and has been developed in the CORDS-based *x*-kernel framework provided by OpenGroup. The tool is implemented as an *x*-kernel protocol layer which can be placed at any level in an *x*-kernel protocol stack. This tool has been used to perform experiments on both the Group Interprocess Communication (GIPC) services from OpenGroup, and middleware and real-time channel services developed as part of the ARMADA project. Further details on ORCHESTRA can be found in several recent reports, e.g., [24, 25].

7.2 COGENT: COntrolled GEneration of Network Traffic

In order to demonstrate the utility of the ARMADA services, it is necessary to evaluate them under a range of operating conditions. Because many of the protocols developed rely on the communication subsystem, it is important to evaluate them under a range of realistic background traffic. Generating such traffic is fairly difficult since traffic characteristics can vary widely depending on the environment in which these services are deployed. To this end, we've developed COGENT (COntrolled GEneration of Network Traffic). COGENT is a networked synthetic workload generator for evaluating system and network performance in a controlled, reproducible fashion. It is based on a simple client-server model and allows the user to flexibly model network sources in order to evaluate various aspects of network and distributed computing.

Implemented in C++ with a `lex/yacc` front end, the current version of the tool takes a high level specification of the distributed workload and generates highly portable C++ code for all of the clients and servers specified. The user can select from a number of distributions which have been used to model a variety of network sources such as Poisson [26, 27], Pareto [28, 29, 30, 26], Log Normal [26], and Log Extreme [27]. The tool then generates the necessary compilation and distribution scripts for building and running the distributed workload.

COGENT is currently being rewritten in JAVA. Both the generator and the generated code will be JAVA based. Because of the portability of JAVA, this will simplify both the compilation and distribution of the workload considerably. We also plan on addressing CPU issues in order to model common activities at the end hosts as well. Another feature being added is the ability for a client or a server to be run in trace-driven mode. That is, to run from a web server or a `tcpdump` [31] log file. Finally, we will be implementing additional source models in order to keep up with the current literature.

8 Conclusions

This paper presented the architecture and current status of the ARMADA project conducted at the University of Michigan in collaboration with the Honeywell Technology Center. We described a number of communication and middleware services developed in the context of this project, and illustrated the general methodology adopted to design and integrate these services. For modularity and composability, ARMADA middleware was realized as a set of servers on top of a microkernel-based operating system. Special attention was given to the communication subsystem since it is a common resource to middleware services developed. We proposed a general architecture for QoS sensitive communication, and also described a communication service that implements this architecture.

We are currently redesigning an existing command and control application to benefit from ARMADA middleware. The application requires bounded time end-to-end communication delays guaranteed by our communication subsystem, as well as fault-tolerant replication and backup services provided by our RTCAST group communication and membership support, and the primary-backup replication service. Testing tools such as ORCHESTRA will help assess communication performance and verify the required communication semantics. Controlled workload generation using COGENT can assist in creating load conditions of interest that may be difficult to exercise via regular operation of the application.

Our services and tools are designed independently of the underlying microkernel or the communication subsystem; our choice of experimentation platform was based largely on the rich protocol development environment provided by *x*-kernel and CORDS. For better portability, we are extending our communication subsystem to provide a socket-like API. We are also investigating the scalability of the services developed. Scaling to large embedded systems may depend on the way the system is constructed from smaller units. We are looking into appropriate ways of defining generic structural system components and composing large architectures from these components such that certain desirable properties are globally preserved. Developing the "tokens" and "operators" of such system composition will enable building predictable analytical and semantic models of larger systems from properties of their individual constituents.

References

[1] N. C. Hutchinson and L. L. Peterson, "The *x*-Kernel: An architecture for implementing network protocols," *IEEE Trans. Software Engineering*, vol. 17, no. 1, pp. 1–13, January 1991.

[2] F.Travostino, E.Menze, and F.Reynolds, "Paths: Programming with system resources in support of real-time distributed applications," in *Proc. IEEE Workshop on Object-Oriented Real-Time Dependable Systems*, February 1996.

[3] A. Mehra, A. Indiresan, and K. Shin, "Structuring communication software for quality of service guarantees," in *Proc. 17th Real-Time Systems Symposium*, pp. 144–154, December 1996.

[4] D. Ferrari and D. C. Verma, "A scheme for real-time channel establishment in wide-area networks," *IEEE Journal on Selected Areas in Communications*, vol. 8, no. 3, pp. 368–379, April 1990.

[5] D. D. Kandlur, K. G. Shin, and D. Ferrari, "Real-time communication in multi-hop networks," *IEEE Trans. on Parallel and Distributed Systems*, vol. 5, no. 10, pp. 1044–1056, October 1994.

[6] C. M. Aras, J. F. Kurose, D. S. Reeves, and H. Schulzrinne, "Real-time communication in packet-switched networks," *Proceedings of the IEEE*, vol. 82, no. 1, pp. 122–139, January 1994.

[7] A. Mehra, A. Indiresan, and K. Shin, "Resource management for real-time communication: Making theory meet practice," in *Proc. 2nd Real-Time Technology and Applications Symposium*, pp. 130–138, June 1996.

[8] R. L. Cruz, *A Calculus for Network Delay and a Note on Topologies of Interconnection Networks*, PhD thesis, University of Illinois at Urbana-Champaign, July 1987. available as technical report UILU-ENG-87-2246.

[9] D. P. Anderson, S. Y. Tzou, R. Wahbe, R. Govindan, and M. Andrews, "Support for continuous media in the DASH system," in *Proc. Int'l Conf. on Distributed Computing Systems*, pp. 54–61, 1990.

[10] A. Mehra, A. Shaikh, T. Abdelzaher, Z. Wang, and K. Shin, "Realizing guaranteed-qos communication services on a micro-kernel operating system," In preparation, July 1997.

[11] K. P. Birman, "The process group approach to reliable distributed computing," *Communications of the ACM*, vol. 36, no. 12, pp. 37–53, December 1993.

[12] Y. Amir, D. Dolev, S. Kramer, and D. Malki, "Transis: A communication sub-system for high availability," Technical Report TR CS91-13, Dept. of Computer Science, Hebrew University, April 1992.

[13] R. van Renesse, T. Hickey, and K. Birman, "Design and performance of Horus: A lightweight group communications system," Technical Report TR94-1442, Dept. of Computer Science, Cornell University, August 1994.

[14] S. Mishra, L. Peterson, and R. Schlichting, "Consul: A communication substrate for fault-tolerant distributed programs," *Distributed Systems Engineering Journal*, vol. 1, no. 2, pp. 87–103, December 1993.

[15] H. Kopetz and G. Grünsteidl, "TTP – a protocol for fault-tolerant real-time systems," *IEEE Computer*, vol. 27, no. 1, pp. 14–23, January 1994.

[16] Y. Amir, L. Moser, P. Melliar-Smith, D. Agarwal, and P. Ciarfella, "The Totem single-ring ordering and membership protocol," *ACM Transactions on Computer Systems*, vol. 13, no. 4, pp. 311–342, November 1995.

[17] F. Cristian, B. Dancy, and J. Dehn, "Fault-tolerance in the advanced automation system," in *Proc. of Fault-Tolerant Computing Symposium*, pp. 6–17, June 1990.

[18] T. Abdelzaher, A. Shaikh, F. Jahanian, and K. Shin, "RTCAST: Lightweight multicast for real-time process groups," in *Proc. IEEE Real-Time Technology and Applications Symposium (RTAS '96)*, pp. 250–259, Boston, MA, June 1996.

[19] S. Johnson. *RTCAST.* Tech report, 1997.

[20] S. Johnson, F. Jahanian, A. Miyoshi, D. de Niz, and R. Rajkumar, "Constructing real-time group communication middleware using the resource kernel," in *Proceedings of the 21st Real-Time Systems Symposium*, 2000.

[21] S. Johnson, F. Jahanian, and J. Shah, "The inter-group router approach to scalable group composition," in *Proceedings of the International Conference on Distributed Computing Systems*, 1999.

[22] H. Zou, "Maintaining temporal consistency in primary-backup replicated real-time data servers," EECS682 Course Project, 1996.

[23] J. Arlat, M. Aguera, Y. Crouzet, J.-C. Fabre, E. Martins, and D. Powell, "Experimental evaluation of the fault tolerance of an atomic multicast system," *IEEE Trans. Reliability*, vol. 39, no. 4, pp. 455–467, October 1990.

[24] S. Dawson, F. Jahanian, and T. Mitton, "Testing of Fault-Tolerant and Real-Time Distributed Systems via Protocol Fault Injection," in *International Symposium on Fault-Tolerant Computing*, pp. 404–414, Sendai, Japan, June 1996.

[25] S. Dawson, F. Jahanian, and T. Mitton, "Experiments on six commercial tcp implementations using a software fault injection tool," to appear in *Software Practice & Experience.*

[26] V. Paxson and S. Floyd, "Wide-area traffic: The failure of poisson modeling," in *SIGCOMM '94*, pp. 257–268, August 1994.

[27] V. Paxson, "Empirically-derived analytic models of wide-area tcp connections," *IEEE/ACM Transactions on Networking*, vol. 2, no. 4, pp. 316–336, August 1994.

[28] W. Leland, M. S. Taqqu, W. Willinger, and D. Wilson, "On the self-similar nature of ethernet traffic (extended version)," *IEEE/ACM Transactions on Networking*, vol. 2, no. 1, pp. 1–15, February 1994.

[29] M. Crovella and A. Bestavros, "Self-similarity in world wide web traffic: Evidence and possible causes," in *SIGMETRICS '96*, May 1996.

[30] M. Garrett and W. Willinger, "Analysis, modeling and generation of self-similar vbr video traffic," in *SIGCOMM '94*, pp. 269–280, 1994.

[31] S. McCanne and V. Jacobso, "The bsd packet filter: A new architecture for user-level packet capture," in *Proceedings of the 1993 Winter USENIX Technical Conference*, San Diego, CA, January 1993.

MISSION
OF
AFRL/INFORMATION DIRECTORATE (IF)

The advancement and application of Information Systems Science and Technology to meet Air Force unique requirements for Information Dominance and its transition to aerospace systems to meet Air Force needs.